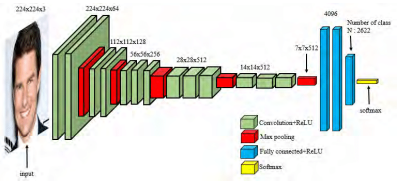
Deep Learning
for generative modeling

Machine Learning 2020
mlvu.github.io

---

tip 1: transfer learning

```
# build the VGG16 network
model = applications.VGG16(weights='imagenet', include_top=False)
print('Model loaded.')
```



2

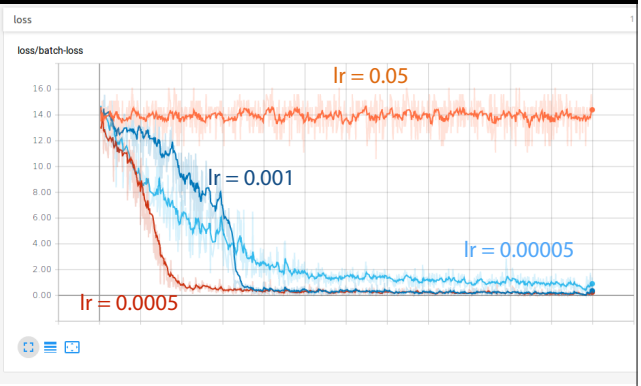
A quick tip for people doing deep learning projects. You can download big models that others have built and trained and re-use them fun your own code. This makes it possible to use very powerful feature detectors, without training for ages on huge amounts of data.

source: **https://www.researchgate.net/publication/318701491_Forensic_Sketches_Recognition_Using_Deep_Convolutional_Neural_Network/figures?lo=1**

see also: **https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html**

---

tip 2: look at your loss curves



3

A very important tool in deep learning is the loss curve. It plots your loss (on the training or validation set) for every training iteration.

The loss curve shows you whether your model is converging, how muchness it is experiencing and whether that noise is diminishing as the model converges.

Here we see loss curves for the MNIST classification problem from the worksheet for four different learning rates. The curves are shown in a piece of software called tensorboard, which is useful for monitoring learning progress. Ideally, you'd print your training loss every batch and your validation loss every epoch.

We see here that for the learning rates 0.05 and 0.00005, the model doesn't learn as well as for the others. 0.0005 seem to provide the optimal balance between fast convergence, and a good minimum.

the plan

part 1:
Generators

Generative Adversarial Networks

part 2:
Autoencoders
dimensionality reduction

Variational Autoencoders
generative models

4

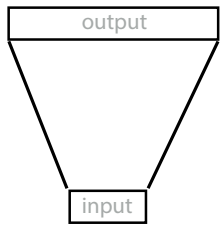
generative models

We now turn to the main topic of today: generative modeling. Generative modeling aims to create a model of the distribution that generated the data, in such a way that we can sample from that distribution. In this example, the dataset is a large collection of images of faces. The aim, then, is to construct a probability distribution over the set of all images from which we can sample, and which makes images of people (or rather, images that look like they're of people) likely.

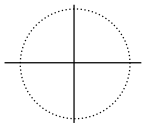
These are two samples from such a model (specifcally, a GAN).

*source: **A Style-Based Generator Architecture for Generative Adversarial Networks**, Karras et al.*

visual shorthand



any feedforward
neural network

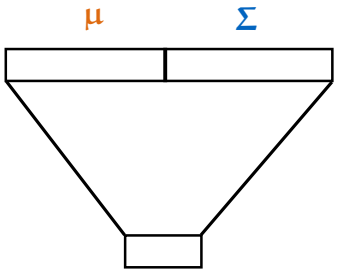


a (standard)
multivariate normal distribution

6

how to turn a NN into a probability distribution

option 1:

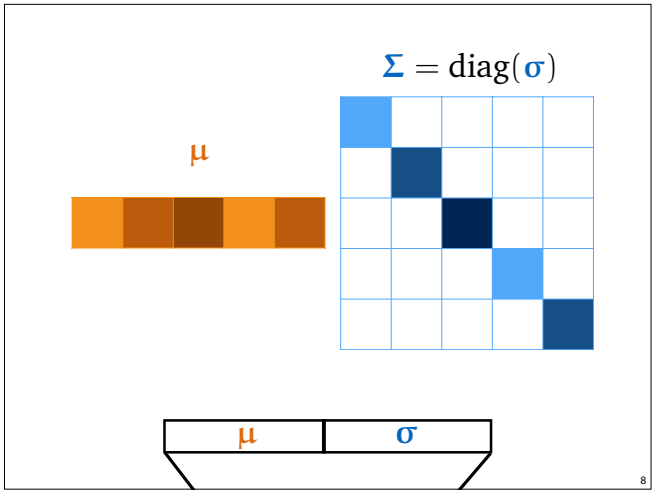


7

Of course, a plain neural network is purely deterministic. It translates an input to an output and does the same thing every time. How to we turn this into a probability distribution?

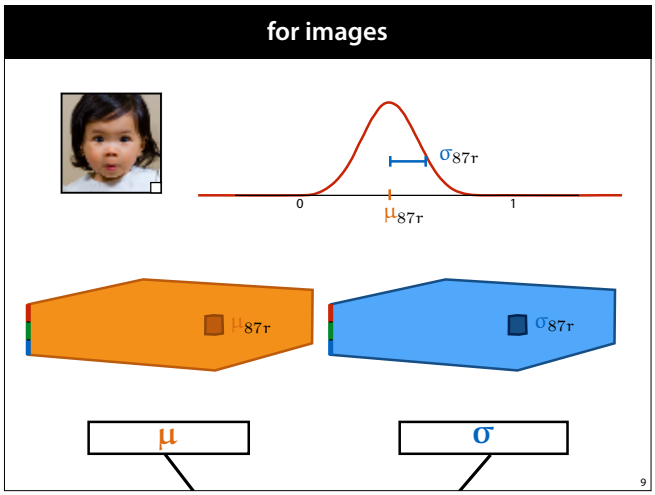
The first option is to take the output and to interpret it as the parameters of a multivariate normal (a μ and a Σ). If the dimensionality of the output is large, we'll often take Σ to be a diagonal matrix (which is why it's the same size as μ in this image).

This doesn't produce very interesting distributions (it's always just an MVN), but it does allow the network to communicate how sure it is about the output: the smaller the variances in Σ, the surer it is about its output. This has the effect of smoothing the loss surface, which can help training a lot. The advantage of this method is that it both allows us to sample from the generator and to compute the probability density $p(x|input)$



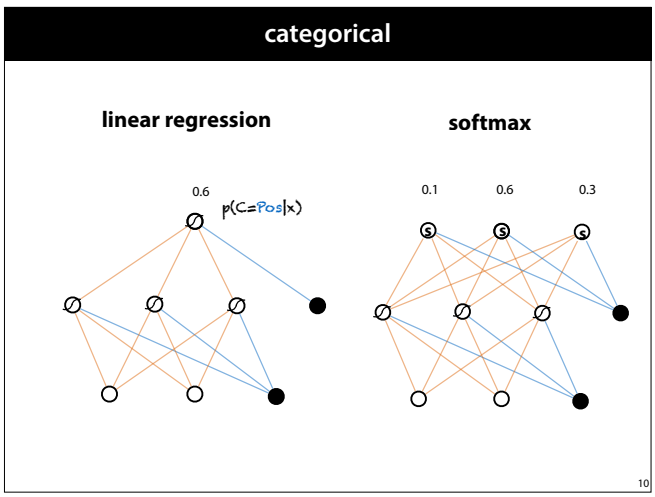
Since representing a full covariance matrix would grow very big, we usually assume that the covariance matrix only has nonzero values on the diagonal. That way the representation of the covariance requires as many arguments as the representations of the mean.

Equivalently, we can think of the output distribution as putting an independent 1D Gaussian on each dimension.

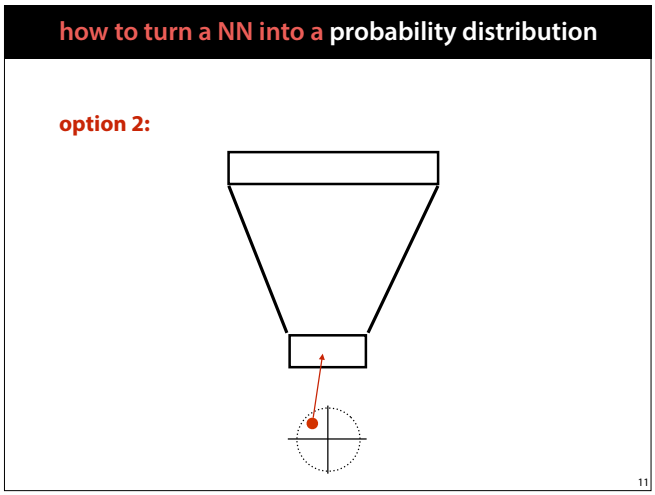


Here's what that looks like when we generate an image. The output distribution give us a mean value for every channel of every pixel (a 3-tensor) and a corresponding variance for every mean.

If we know what the output should be, we can optimise the parameters to maximise the likelihood of the true value.

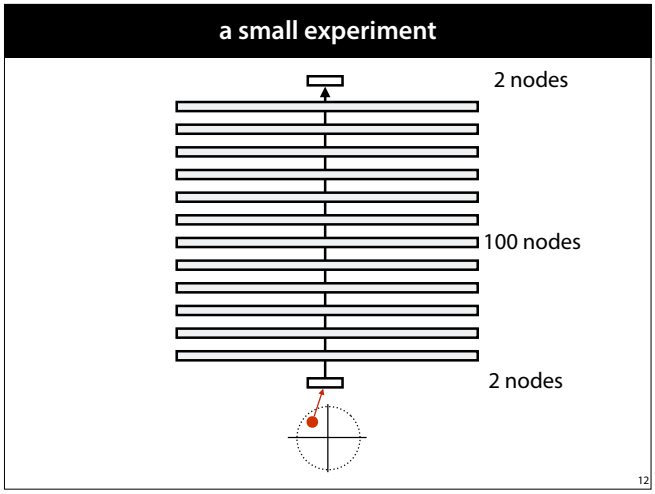


This principle is not new. In neural network classification, the standard approaches we've seen (a linear regression layer and a softmax output) the neural network also produces the parameters for a probability distribution on the output space (a Bernoulli distribution on the left and a Categorical on the right).



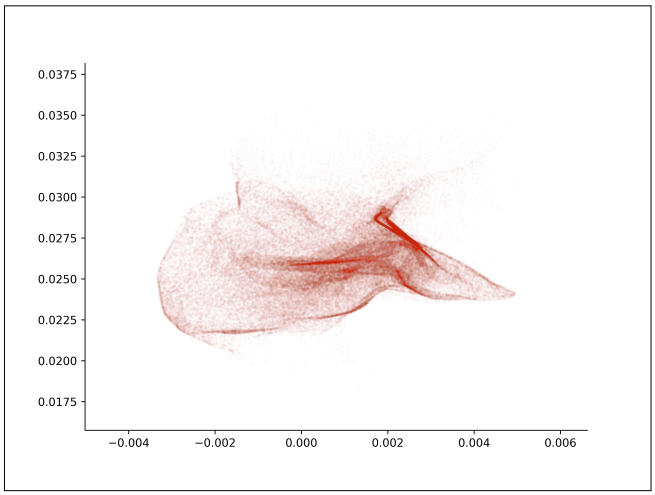
A more exciting option is to keep the output deterministic, but *sample* the input from a standard MVN, and feed it to the neural network. In other words, we let the neural network transform the standard MVN.

For this option, we cannot easily compute the probability density for a given instance, but we can easily sample.



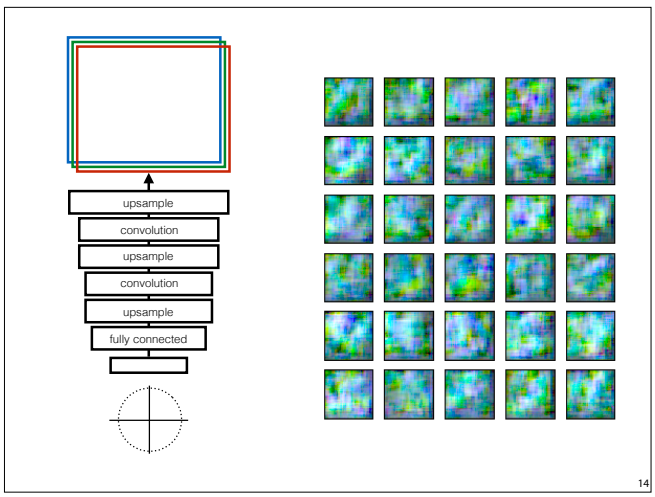
To see what kind of distributions we might learn from option 2, let's try a little experiment. We wire up a random network as shown: a two-node input layer, followed by 12 100-node fully-connected hidden layers with ReLU activations.

We *don't train* the network. We just use Glorot initialisation to pick the parameters, and sample some points.



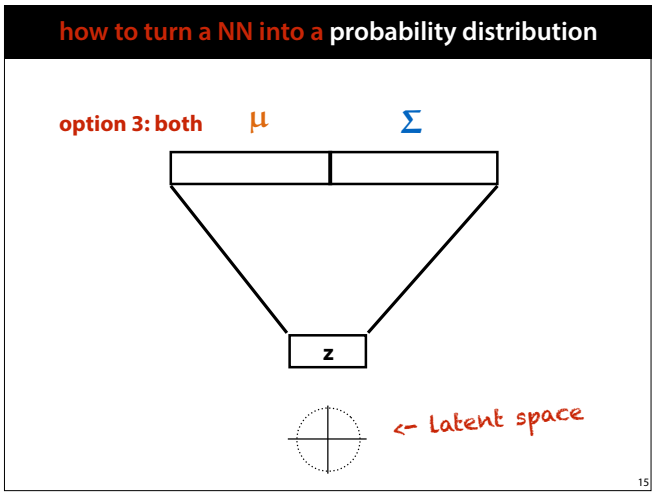
Here's a plot of 100k points sampled in this way. Clearly, these are highly complex distributions (remember, this network hasn't even been trained).

Note also that the variance has shrunk, and the mean has drifted away from (0, 0); apparently the weight initialisation is not quite perfect.



As noted before, the actual network can be anything. Here is what we get if we make the output a 3-tensor representing an image, and we make the hidden layers convolutions and upsampling layers. To the right are 30 samples from this network.

Note: I've enhanced the contrast and saturation to ensure that the colors show up on a projector.



Of course, we can also do both: we sample the input from a standard MVN, interpret the output as another MVN, and then sample from that.

In these kinds of generator networks, the input is often called **z**, and the space of inputs is often called the **latent space**.

training a generator: the naive approach

loop:

Generate a random output **y**

Sample a random instance **x**

Compute loss(**x**, **y**) and backpropagate

loss: mean-squared error, binary cross-entropy, L1, etc.

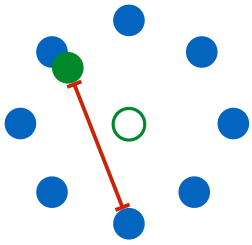
16

How do we train models like this? Here is a naive approach: we simply sample a random point (e.g. a picture) from the data, and sample a point from the model and train on how close they are.

The loss could be any distance function between two tensors. The mean-squared error over the elements is a simple approach. If the elements are values between 0 and 1 (like in images), the binary cross-entropy makes sense too. For images the absolute value of the error (also called L1 loss) is also popular.

---

mode collapse



17

If we implement the naive approach, we get something called *mode collapse*. Here is an example: the blue points represent the modes (likely points) of the data. The green point is generated by the model. It's close to one of the blue points, so the model should be rewarded, but it's also far away from some of the other points. During training, there's no guarantee that we pair it up with the correct point, and we are likely to compute the loss to a completely different point.

On average the model is punished for generating such points as much as it is rewarded. The model that generates only the open point in the middle gets the same loss (and less variance). Under backpropagation, neural networks tend to converge to a distribution that generates only the open point over and over again.

In other words, the many different modes (areas of high probability) of the data distribution end up being averaged ("collapsing") into a single point.

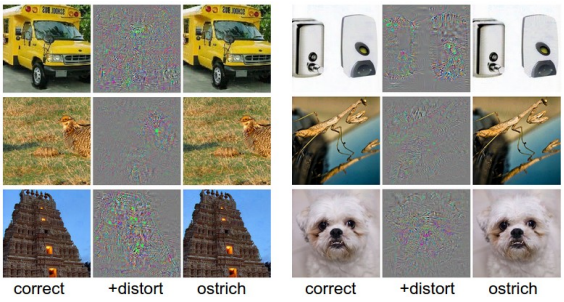
---

mode collapse

18

We want the network to *imagine details*, not to average over all possibilities.

adversarial examples (2014)



source: **http://karpathy.github.io/2015/03/30/breaking-convnets/**

19

In order to avoid mode collapse we need to come up with better ways of training these models. The first we will investigate is called "generative adversarial networks."

They originated just after ConvNets were breaking new ground, showing spectacular, sometimes super-human, performance in image labeling.

Researchers decided to start investigating what kind of inputs would make a trained ConvNet give a certain output. This is easy to do, you just compute the gradient with respect to the input of the network, and train the input to maximise the activation of a particular label. They ended up with simple noisy images that looked nothing like the required label to us.

Moreover, they found that if they started the search at an image of another class, all that was needed to turn it into another class was a very small distortion. So small, that to us the image looked unchanged. These kinds of manipulated inputs are called **adversarial examples**, and they're an active area of research (both how to generate them and make models more robust against them).

---

generative adversarial networks

loop:

train a classifier to tell X_{Pos} from X_{Neg}

Generate adversarial examples
Clearly not Pos, but the classifier thinks so anyway

Add the adversarial examples to X_{Neg}

The classifier (discriminator) gets more *robust*, the generator gets more *realistic*.

20

Pretty soon, this bad news was turned into good news by realising that if you can generate adversarial examples automatically, you can also add them to the dataset (as negatives) and retrain your network to make it more robust.

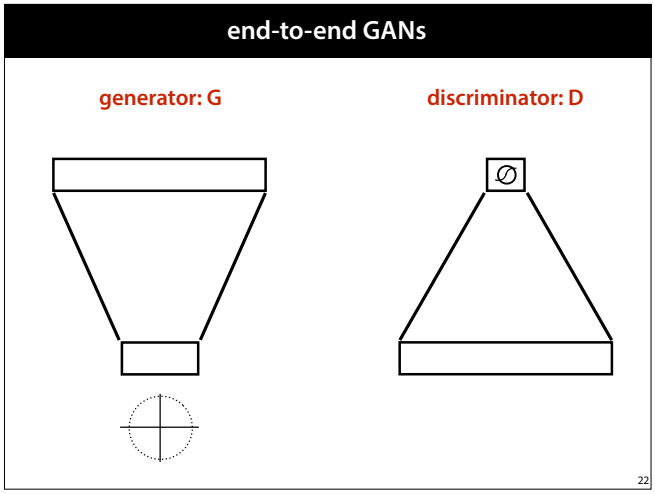
We can think of this as a kind of iterated 2 player game (or an arms race). The Generator (the process that generates the adversarial examples) tries to get good enough to fool the classifier and the classifier tries to get good enough to tell the fakes from the true examples.

GANs

- Vanilla GANs
- Conditional GANs
- CycleGAN
- StyleGAN

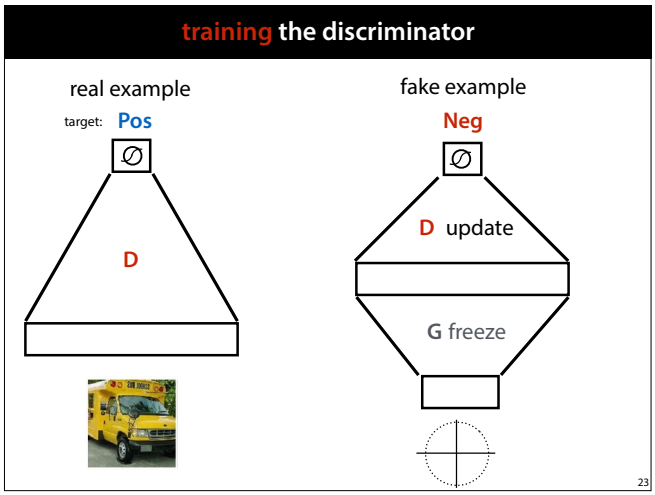
21

We'll look at four different examples of GANs



We will draw the two components like this. The **generator** takes an input sampled from a standard MVN and produces an image. The **discriminator** takes an image and classifies it as Pos (a real image) or Neg (a fake image sampled from the generator).

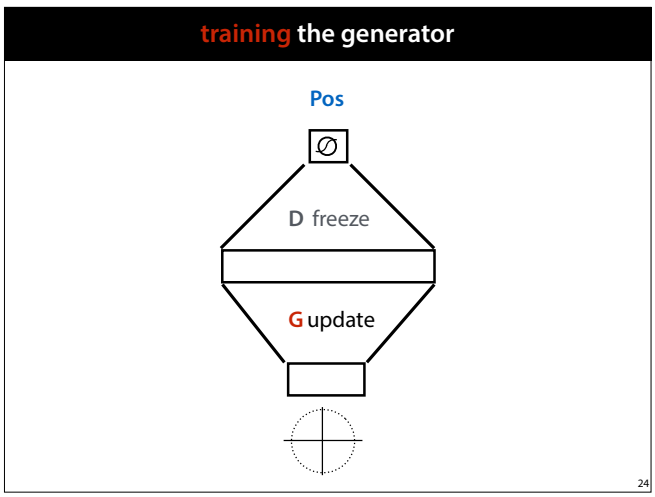
If we have other images that are not of the target class, we can add those to the negative examples as well, but often, the positive class is just our dataset (like a collection of human faces), and the negative class is just the fake images created by the generator.



To train the discriminator, we feed it examples from the positive class, and train it to classify these as Pos. Note that we're not forcing it, we're just giving the weights a small nudge, through backpropagation, to make Pos more likely for this example.

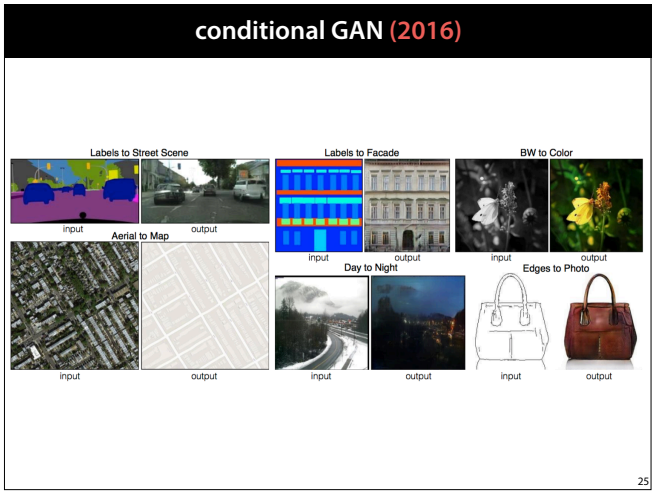
We also sample images from the generator (whose weights we don't update) and train the discriminator to make these negative.

At first these will just be random noise, but there's little harm in telling our network that such images are not busses (or whatever our positive class is).



Then, to train the generator, we freeze the discriminator and train the weights of the generator to produce images that cause the discriminator to label them as **Positive**.

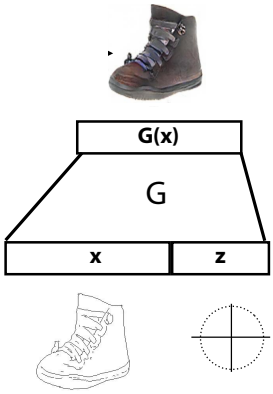
We can alternate these steps every batch, we don't have to wait for either model to converge.



Sometimes we want to train the network to take an input, but to generate the output probabilstically. Here, the network needs to fill in realistic details. For instance, many colors are allowed for the flower, but the network should pick one, and not average over all colors.

For this purpose, we can use conditional GANs.

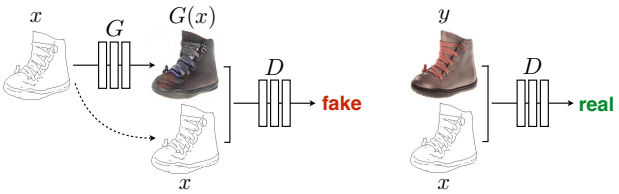
generator is now a function



source: **Image-to-Image Translation with Conditional Adversarial Networks** (2016), Phillip Isola Jun-Yan Zhu et al. 26

In a conditional GAN, the generator is a function with an image input, which it maps it to an image output. However, it uses randomness to imagine specific details in the output. running this generator twice would result in different shoes that are both "correct" instantiations of the input line drawing.

cGAN

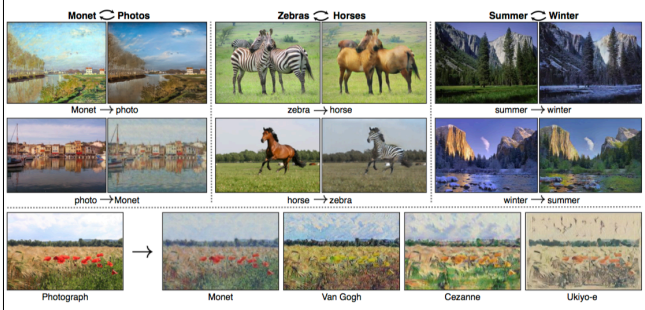


source: **Image-to-Image Translation with Conditional Adversarial Networks** (2016), Phillip Isola Jun-Yan Zhu et al. 27

We feed the discriminator either an input/output pair from the data (right), which it should classify as real, or a an input from the data together with the output generated by the generator, which it should classify as fake.

The generator is trained in two ways. (a) we freeze the weights of the discriminator, as before, and train the generator to produce thins that the discriminator will think are real. (b) We feed it and input from the data, and back propagate on the corresponding output (using L1 loss).

for unpaired images



source: **Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (2017)** Zhu et al 28

That only works if the input and output are matched. For some tasks, we only have unmatched bags of images in two domains.

If we randomly match one image in one domain to an image in another domain, we get mode collapse again.

CycleGAN (2017)

Transformations are always learned both ways.
Ingredients:

- Horse discriminator
- Zebra discriminator
- Horse-to-zebra generator (G)
- Zebra-to-horse generator (F)

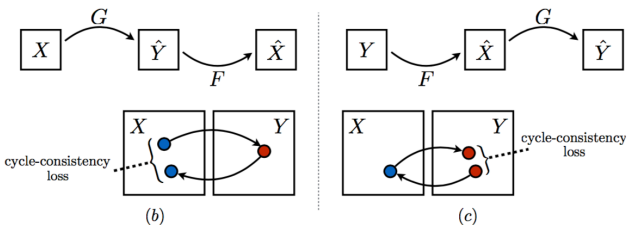
29

CycleGANs achieve this by adding a "cycle consistency term" to the loss function. For instance, in the horse-to-zebra example, if we transform a horse to a zebra and back again, the result should be close to the original image. Thus, the objective becomes to train a horse-to-zebra transformer and a zebra-to-horse transformer together in such a way that:

- a horse-discriminator can't tell the generated horses from real ones (and likewise for a zebra discriminator)
- the cycle consistency loss for both combined is low.

After some training of the generators, the discriminators are retrained with with the original data and some generated horses and zebras.

CycleGAN (2017)



Think of the generators as practicing *steganography*.

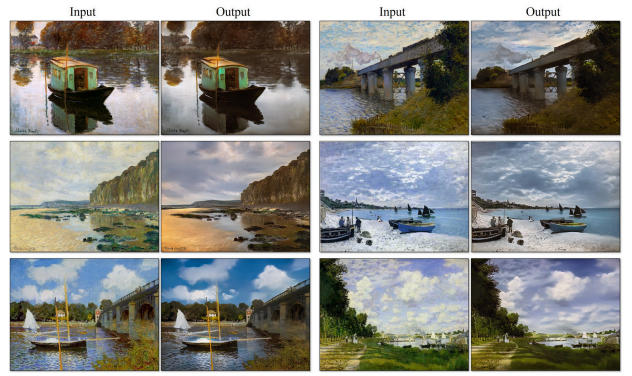
I.e. hiding a horse picture inside a zebra picture.

source: **Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks (2017)** Zhu et al 30

CycleGANs achieve this by adding a "cycle consistency term" to the loss function. For instance, in the horse-to-zebra example, if we transform a horse to a zebra and back again, the result should be close to the original image. Thus, the objective becomes to train a horse-to-zebra transformer and a zebra-to-horse transformer together in such a way that:

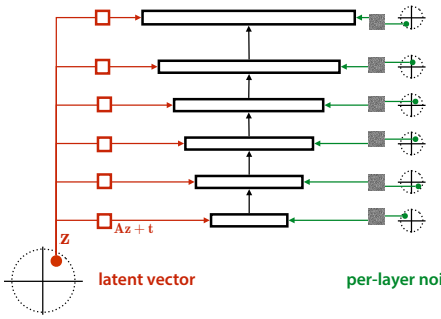
- a horse-discriminator can't tell the generated horses from real ones (and likewise for a zebra discriminator)
- the cycle consistency loss for both combined is low.

After some training of the generators, the discriminators are retrained with with the original data and some generated horses and zebras.



31

StyleGAN (2018)



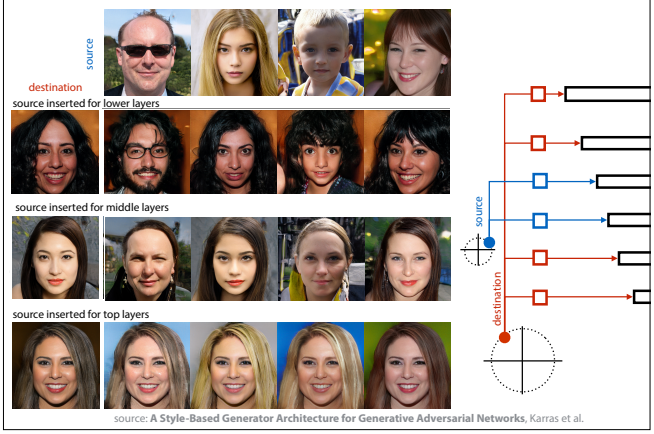
source: **A Style-Based Generator Architecture for Generative Adversarial Networks**, Karras et al. 32

Many other tricks (a pre-generator mapping network, Wasserstein loss, progressive growing) were used, but this is the idea we'll focus on here: to feed the network the later vector at each layer.

Since a deconvolution starts with a coarse (low resolutiom), high level description of an image, and slowly fills in the details, feeding it the latent vector at every layer (transformed by an affine transformation to fit it to the shape of the data at that stage), allows it to use different parts of the latent vector to describe different aspects of the image (the authors call these "styles").

The network also receives separate extra random noise per layer, that allows it to make random choices. Without this, all randomness would have to come from the latent vector.

changing the latent vector



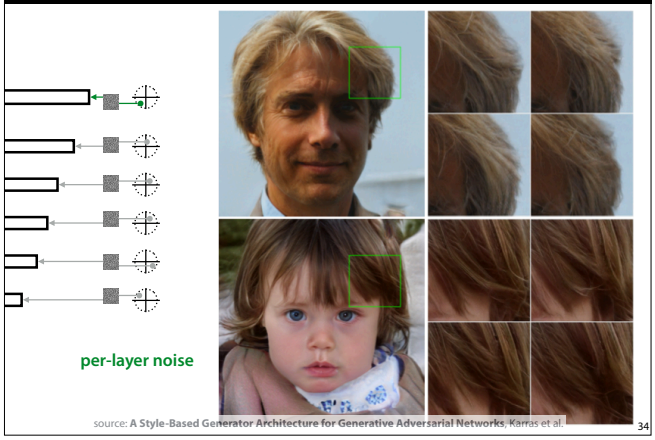
source: **A Style-Based Generator Architecture for Generative Adversarial Networks**, Karras et al.

This allows us to manipulate the network (after training), by changing the latent vector to another for some of the layer. In this example all images on the margins are people that are generated for a particular latent vector.

We then generate the image for the destination, except that for a few layers (at the bottom, middle or top), we use the source latent vector instead. This principle was used in training as well as after: the discriminator was fed samples from a single z vector and from this kind of mixture of two latent vectors.

As we see, overriding the bottom layers changes things like gender, age and hair length, but not ethnicity. For the middle layer, the age is largely taken from the destination image, but the ethnicity is now override by the source. Finally for the top layers, only surface details are changed.

changing the noise



source: **A Style-Based Generator Architecture for Generative Adversarial Networks**, Karras et al.

34

If we keep all the noise inputs the same except for the very last one, we can see what the noise achieves: the man's hair is equally messy in each generated example, but exactly in what way it's messy changes per sample. The network uses the noise to determine the precise orientation of the individual "hairs".

GANs: not discussed

- Wasserstein distance
- Evaluation: Inception score, FID score
- Batch normalisation
- Relativistic GANs

35

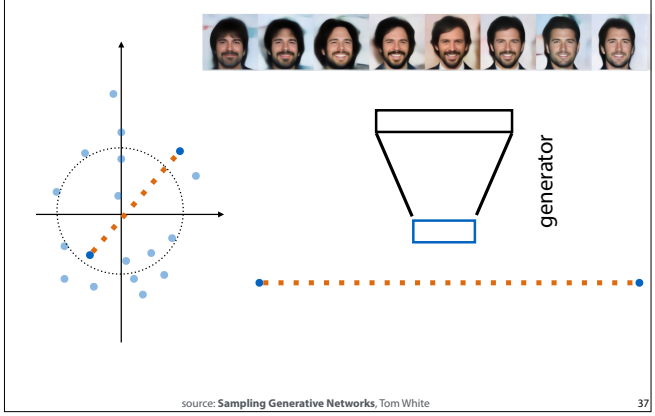
We've given you a high level overview of GANs, which will hopefully give you an intuitive grasp of how they work. However, GANs are notoriously difficult to train, and many other tricks are required to get them to work. Here are some phrases you should Google if you decide to try implementing your own GAN.

What can we do with a generator?

- Generate "new" data
- Interpolation
- Data manipulation
  autoencoders only
- Dimensionality reduction
  autoencoders only

36

interpolation



source: **Sampling Generative Networks**, Tom White

37

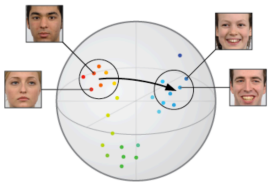
If we take two points in the latent space, and draw a line between them, we can pick evenly spaced points on that line and decode them. If the generator is good, this should give us a smooth transition from one point to the other, and each point should result in a convincing example of our output domain.

interpolation grid

source: **Sampling Generative Networks**, Tom White

We can also draw an interpolation grid; we just map the corners of a square lattice of equally spaced points to four points in our latent space.

spherical linear interpolation



source: **Sampling Generative Networks**, Tom White

If the latent space is high dimensional, most of the probability of the standard MVN is near the edges of the radius-1 hypersphere (not in the centre as it is in 1, 2 and 3-dimensional MVNs). High-dimensional MVNs look more like a soap bubble than the dense pointcloud we're used to seeing in low-dimensional visualizations.

For that reason, we get better results if we interpolate along an arc instead of along a straight line. This is called **spherical linear interpolation**.

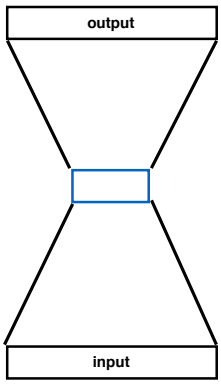
What can we do with a generator?

- Generative modelling
- Interpolation
- Data manipulation
  autoencoders only
- Dimensionality reduction
  autoencoders only

requires mapping into the latent space

break

autoencoders



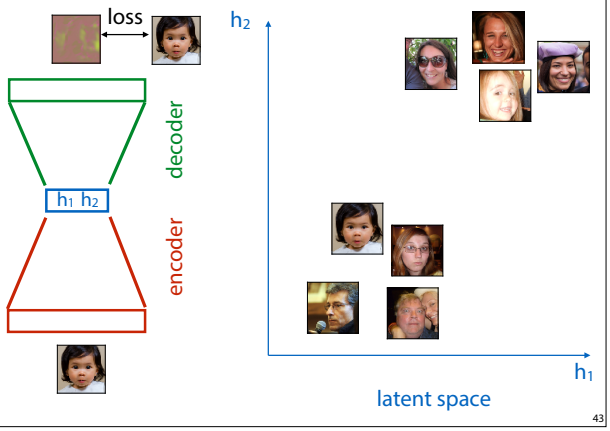
bottleneck architecture for dimensionality reduction.

input should be as close as possible to the output

but: must pass through a small representation.

42

An **autoencoder** is a particular type of neural network, shaped like an hourglass. Its job is just to make the output as close to the input as possible, but somewhere in the network there is a small layer that functions as a bottleneck. After the network is trained, this small layer becomes a compressed representation of the input.



43

Here we call the blue layer the latent representation of the input. If train an autoencoder with just two nodes on the latent representation, we can plot what latent representation each input is assigned. If the autoencoder works well, we expect to see similar images clustered together (for instance smiling people vs frowning people, men vs women, etc).

In a 2D space, we can't cluster too many attributes together, but in higher dimensions it's easier. To quote **Geoff Hinton**: "If there was a 30 dimensional supermarket, [the anchovies] could be close to the pizza toppings and close to the sardines."

---

after 5 epochs (256 hidden units)

44

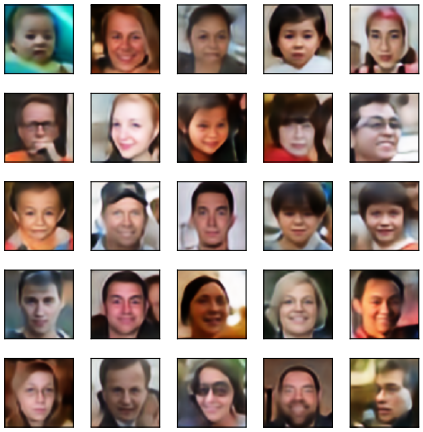
Here are the results on a very simple network (just 4 fully connected ReLU layers for the encoder and the decoder), with MSE loss on the output.

---

after 25 epochs

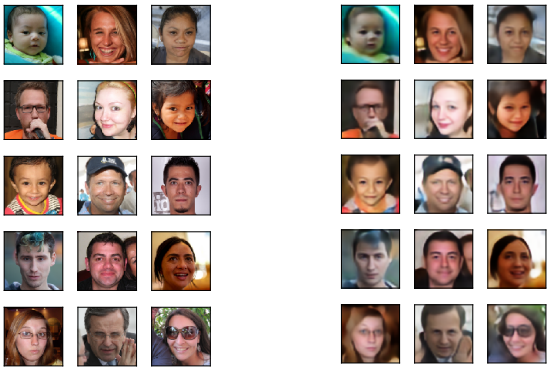
45

after 100 epochs



46

after 300 epochs

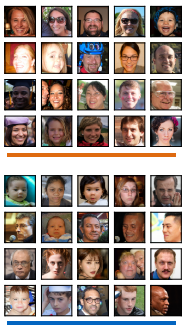


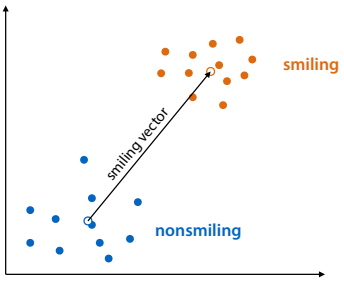
data

reconstructions

47

find the smiling vector





latent space (256 dim)

48

To find the direction in latent space that we can use to make people smile, we just label a small amount of instances as smiling and nonsmiling, and draw the vector between their respective means.

This is one big benefit of autoencoders: we can train them on unlabelled data (which is cheap) and then use only a small number of labeled examples to "annotate" the latent space.

make someone smile/frown

encode to the latent space:

$z = \text{encode}(x)$

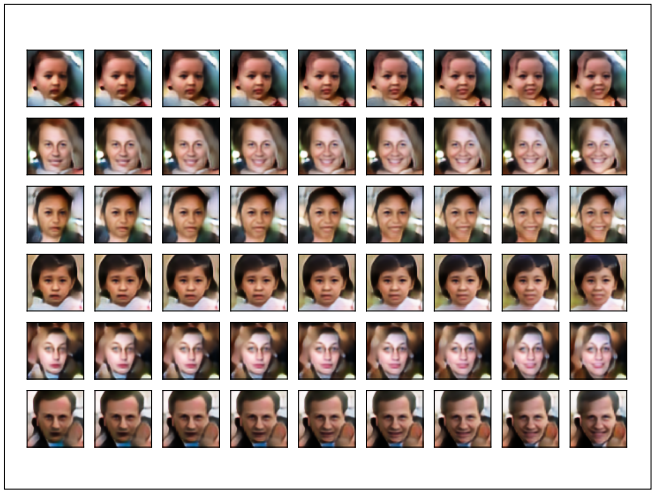
add/subtract some proportion of the smiling vector:

$z_{smile} = z + v_{smile} * 0.2$

decode to a smiling face:

$x_{smile} = \text{decode}(z_{smile})$

49



turning an autoencoder into a generator

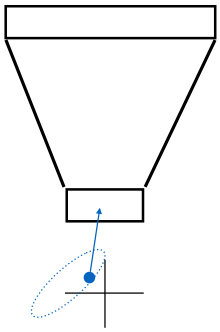
train an autoencoder

encode the data to latent variables Z

fit an MVN to Z

sample from the MVN

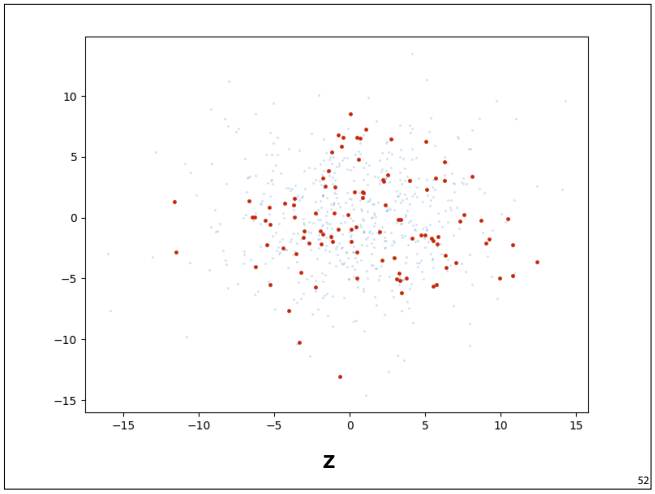
"decode" the sample



51

But this lecture was about generative models. Once we've trained an autoencoder to map faces to a latent dimension, can we turn it into a generator for human faces?

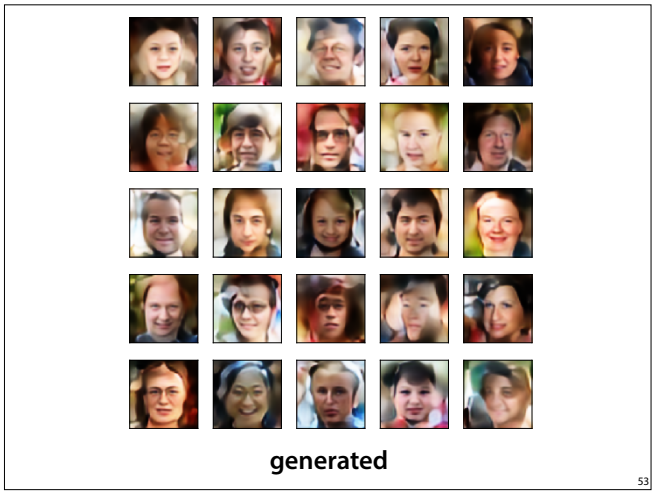
Here's a simple algorithm to get a generative model from an autoencoder.



52

This is the point cloud of the latent representations. We plot the first two of the 128 dimensions.

In this 128D space, we fit an MVN, and sample 400 points from it (the red dots).

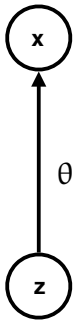


53

If we feed these points to the decoder, this is what we get. This is a decent enough way of turning an autoencoder into a generative model. But it's not very *principled*: we're training for reconstruction error, and then turning the result into an autoencoder. Is there a way to train for **maximum likelihood** directly?

variational autoencoders

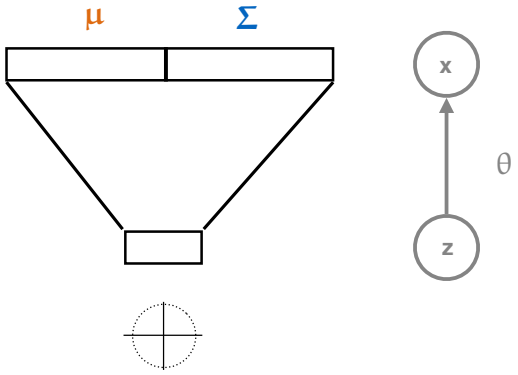
- Force the decoder to also decode points near z correctly
- Forces the latent distribution of the data towards N(**0**, **I**)
- Can be derived from first principles
  maximum likelihood



At heart, the generator model we sketched before the break (option 2) is a *hidden variable* model. We sample some **z**, pass it through the network and observe the output **x**. If we knew the **z** for each **x**, our job would be a lot easier. But we don't.

We ran into a similar situation with the EM algorithm for Gaussian Mixture models. Let's review that model, and see if it can provide some inspiration.

hidden variable model



We now have the tools to start applying these principles to the neural network generative model. We start with our standard probability distribution built around a neural net.

The first problem that we run into, is that with this model, it's very difficult to compute p(z|x,θ) explicitly even if we know θ (this was easy for the Gaussian mixture model, but not here).

Instead, we will approximate p(z|x, θ) with a neural network, and make that our q function.

maximum likelihood objective

$$\arg\max_{\theta} \quad \ln p(x \mid \theta)$$

θWe'll start with this objective. We want to choose our parameters θ (the weights of the neural network) to maximise the log likelihood of the data. We will write this objective step by step until we end up with an autoencoder.

a very useful decomposition

$$\ln p(\mathbf{x} \mid \theta) = L(q, \theta) + KL(q, p)$$

with :

$p = p(\mathbf{z} \mid \mathbf{x}, \theta)$

$q(\mathbf{z} \mid \mathbf{x})$ any approximation to $p(\mathbf{z} \mid \mathbf{x})$

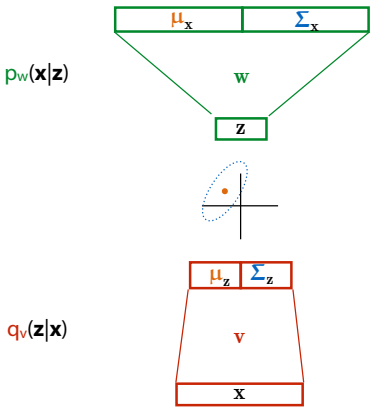
$KL(q, p)$ Kullback-Leibler divergence

$$L(q, \theta) = \mathbb{E}_q \ln \frac{p(\mathbf{x}, \mathbf{z} \mid \theta)}{q(\mathbf{z} \mid \mathbf{x})}$$

58

As with the gaussian mixture model, we cannot marginalize out the hidden variable **z**, or compute the probability over **z** given x. Instead, we use an approximation on the probability of **z** given x, called **q**. and optimise both the probability of **x** given **z** and **z** given **x**.

In the Gaussian mixture model we could pick some choice of parameters θ and them compute the probability on z given those. In this case that would require us to invert the network: given some network weights and an output, which inputs would be most likely to produce that output? This is an expensive and complex operation. We avoid it by modelling q with a second network which maps from **x** to **z** directly. Since this decomposition holds for any q, we can just apply it, and train q together with p.



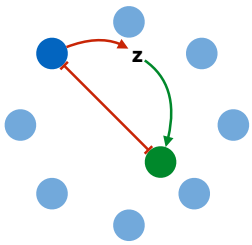
59

This is our model. The neural network p_w is our probability distribution. q_v is our approximation of the conditional distribution on **z**.

If we feed q_v an instance from our data **x**, we get a distribution on **z**. If we sample a point from this distribution, and feed it to p_w we get a distribution on **x**.

---

mode collapse



60

Note how this solves the mode collapse problem. Before, when we had only the generator, we could sample an output, but we didn't know which instance to compare it to. Now we map an input to the latent space and back to the data space, so we know which instance the generated output should look like.

---

simplify our notation

$$p(\mathbf{x} \mid \mathbf{z}, \theta) = p_w(\mathbf{x} \mid \mathbf{z})$$

$$q(\mathbf{z} \mid \mathbf{x}, \theta) = q_v(\mathbf{z} \mid \mathbf{x})$$

61

Since the parameters of our model are the neural network weights, we'll simplify our notation like this. Because these probabilities are described by neural networks, these are the only functions we can efficiently compute. We can't marginalise arguments out, or reverse the conditional.

a very useful decomposition

$$\arg\max_{w} \sum_{x} \ln p_w(x)$$

$$\ln p_w(x) = L(q_v, p_w) + KL(q_v, p_w)$$

with:

$q_v(z \mid x)$: any approximation to $p_w(z \mid x)$

$$L(q_v, p_w) = \mathbb{E}_q \frac{p_w(x, z)}{q_v(z \mid x)}$$

62

Here is the earlier decomposition, rewritten in our new notation.

---

what's our loss?

$$\ln p_w(x) = L(q_v, p_w) + KL(q_v, p_w)$$

variational lower bound
or evidence lower bound (ELBO)

$$L(q_v, p_w) = \mathbb{E}_q \frac{p_w(x, z)}{q_v(z \mid x)}$$

63

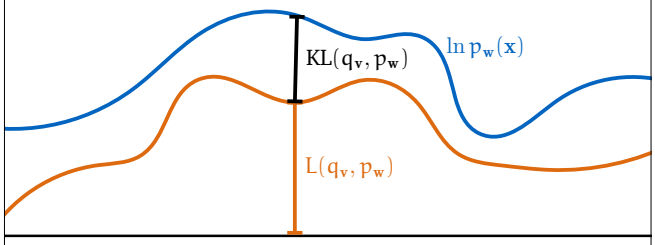
Because we cannot invert p_w, we cannot easily compute the KL term (let alone optimise q_v to minimise it). Instead, we focus entirely on the L term. Since it's a lowerbound on the quantity we're trying to maximize, anything that increases L will help our model. The better we maximise L, the better our model will do.

Right now, we can't use L as a loss function directly it contains functions like p(x,z) that are not easily computed and because it's an expectation. We'll rewrite it step by step until it's a loss function that can de directly cheaply and easily in a deep learning system.

---

lower bound loss

$$\ln p_w(x) = L(q_v, p_w) + KL(q_v, p_w)$$



64

Here's a visualisation of how a lower bound loss works. We're interested in finding the highest point of the blue line (the maximum likelihood solution), but that's difficult to compute. Instead, we maximise the orange line (the evidence lower bound). Because it's guaranteed to be below the blue line everywhere, we may expect to be finding a high value for the blue line as well.

How well we do on the blue line depends a lot on how *tight* the lower bound is. The distance between the lower bound and the log likelihood is expressed by the KL divergence between $p_w(z|x)$ and $q_v(z|x)$. That is, because we cannot easily compute $p_w(z|x)$, we introduced an approximation $q_v(z|x)$. The better this approximation, the lower the KL divergence, and the tighter the lower bound.

---

minimize -L(**v**, **w**)

$$-L(v, w) = -\mathbb{E}_q \ln \frac{p_w(x, z)}{q_v(z \mid x)}$$

$$= -\mathbb{E}_q \ln \frac{p_w(x \mid z) p_w(z)}{q_v(z \mid x)}$$

$$= -\mathbb{E}_q \ln p_w(x \mid z) - \mathbb{E}_q \ln p_w(z) + \mathbb{E}_q \ln q_v(z \mid x)$$

$$= KL(q_v(z \mid x), p_w(z)) - \mathbb{E}_q \ln p_w(x \mid z)$$

$$p_w(z) = N(0, I)$$

65

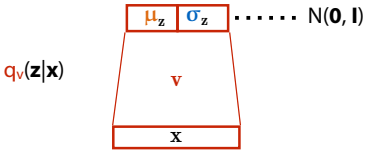
Since we want to implement a loss function, we will *minimize* the negative of the L function. We now need to work out what that means for our neural net.

All three probability functions are known: $q(z|x)$ is the encoder network, $p(x|z)$ is the decoder network, and $p(z)$ was chosen when we defined (back in slide 9) how the generator works, if we ignore x, the distribution on z is a standard multivariate normal.

NB: Since we now take the expectation over a continuous distribution, all sums become integrals and (inside all expectations and inside the KL divergence). By keeping everything written as expectations, we ensure that we don't have to deal with integrals.

loss function

$$\text{loss}(v, w) = \text{KL}(q_v(z \mid x), p_w(z)) - \mathbb{E}_q \ln p_w(x \mid z)$$

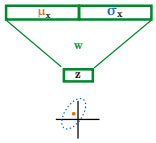


66

The KL term is just the KL divergence between the MVN that the encoder produce for x and the standard normal MVN. This **works out** as a relatively simple differentiable function of mu and sigma, so we can use it directly in a loss function.

Remember that Sigma_z is usually restricted to a diagonal matrix, so the network just outputs a vector of the same size as mu_z, which we take to be the diagonal of the covariance matrix.

$$\text{loss}(v, w) = \text{KL}(q_v(z \mid x), p_w(z)) - \mathbb{E}_q \ln p_w(x \mid z)$$



take L samples $\{z_i\}$ from $q(z|x)$.

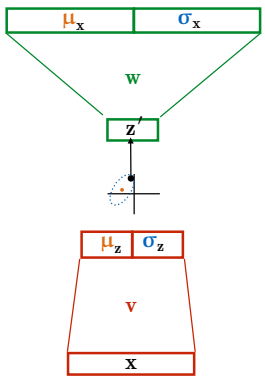
approximate as: $\frac{1}{L}\sum_i \ln p_w(x \mid z_i)$

keep things simple: L=1

67

The second part of our loss function requires a little more work. It's an **expectation** for which we don't have a closed form expression. Instead, we can *approximate* it by taking some samples, and averaging. To keep things simple, we just take a single sample (we'll be computing the network lots of times during training, so overall, we'll be taking lots of samples).

$$\text{loss}(v, w) = \text{KL}(q_v(z \mid x), N(0, I)) - \ln p_w(x \mid z')$$



68

We almost have a fully differentiable model. Unfortunately, we still have a sampling step in the middle (and sampling is not a differentiable operation).

---

sampling

$$N(\mu, \Sigma) \quad : \quad X\sigma + \mu \quad \text{with} \quad X \sim N(0, 1)$$

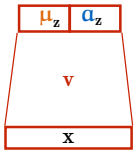
$$N^d(0, 1) \quad : \quad \begin{pmatrix} X_1 \\ \vdots \\ X_d \end{pmatrix} \quad \text{with} \quad X_i \sim N(0, 1)$$

$$N^d(\mu, \Sigma) \; : \; AX + \mu \;\; \text{with} \;\; X \sim N^d(0, 1), \quad \Sigma = AA^T$$

We can get get rid of it by remembering the algorithm for sampling from a given MVN. We sample from the standard MVN, and *transform* the sample using the parameters of the required MVN.

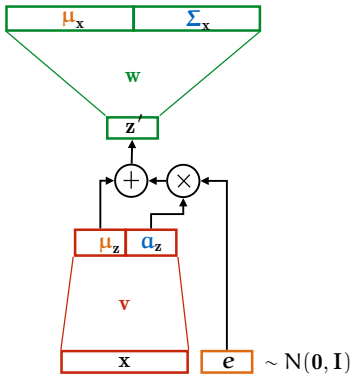
$$\Sigma = AA^T$$

$$\Sigma = \text{diag}(\sigma) = \text{diag}(a^2)$$



70

$$\text{loss}(v, w) = \text{KL}(q_v(z \mid x), N(0, I)) - \ln p_w(x \mid eA_z + \mu_z)$$



71

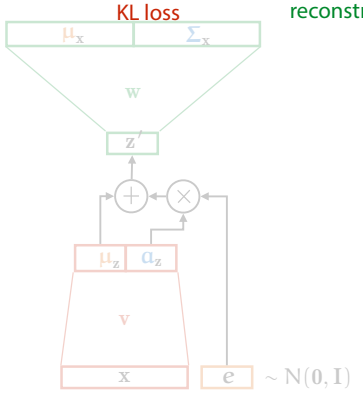
We can work this sampling into the architecture. We provide the network with an **extra input**: a sample from the standard MVN. We also slightly change the q network.

Instead of making the network produce Sigma, we make it produce A, since the network is trained anyway, we can just interpret the output as A instead of Sigma. Since both are diagonal matrices A just contains the square roots of Sigma.

Why does this help us? We're still sampling, but we've moved the random sampling out of the way of the backpropagation. The gradient can now propagate down to the weights of the q function, and the actual randomness is treated as an *input*, rather than a computation.

$$\text{loss}(v, w) = \text{KL}(q_v(z \mid x), N(0, I)) - \ln p_w(x \mid eA_z + \mu_z)$$

KL loss — reconstruction loss



72

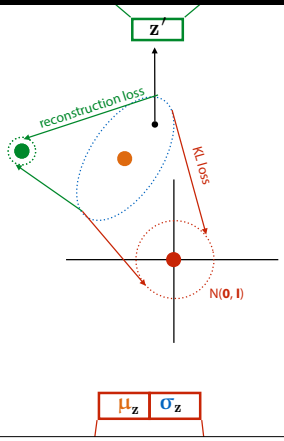
The two terms of the loss function are usually called KL loss and reconstruction loss.

The reconstruction loss maximises the probability of the current instances. This is basically the same loss we used for the regular auto-encoder: we want the output of the decoder to look like the input.

The KL loss ensures that the latent distributions are clustered around the origin, with variance 1. Over the whole dataset, it ensures that the latent distribution looks like a standard normal distribution.

---

three forces



73

The formulation of the VAE has three forces acting on the latent space. The reconstruction loss pulls the latent distribution as much as possible towards a single point that gives the best reconstruction. Meanwhile, the KL loss, pulls the latent distribution (for all points) towards the standard normal distribution, acting as a **regularizer**. Finally, the sampling step ensures that not just a single point returns a good reconstruction, but a whole *neighbourhood* of points does. The effect can be summarized as follows:

The **reconstruction loss** ensures that there are points in the latent space that decode to the data.

The **KL loss** ensures that all these points together are laid out like a standard normal distribution.

The **sampling step** ensure that points in between these also

decode to points that resemble the data.

choosing rec. loss

$$-\ln p_w(\mathbf{x} \mid \boldsymbol{\epsilon} A_z + \mu_z)$$

μ_x | Σ_x

$$-\ln N(x|\mu, \sigma)$$

$$-\ln N(x|\mu, c) \to (x = \mu)^2$$

squared error

$$|x = \mu|$$

absolute error
sharper images
Laplace distribution

$$H(\text{output}, \text{target})$$

cross-entropy
fast conv, obscure distribution

74

The two terms of the loss function are usually called KL loss and reconstruction loss.

The reconstruction loss maximises the probability of the current instances. This is basically the same loss we used for the regular auto-encoder: we want the output of the decoder to look like the input.

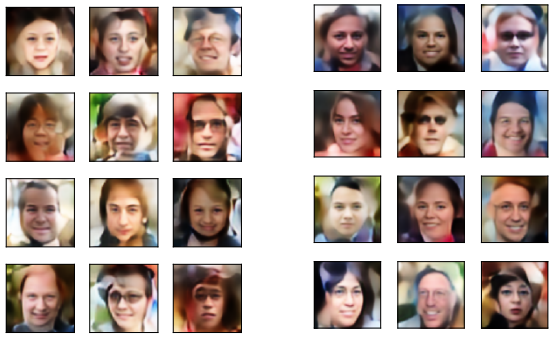
The KL loss ensures that the latent distributions are clustered around the origin, with variance 1. Over the whole dataset, it ensures that the latent distribution looks like a standard normal distribution.

after 300 epochs

data ae vae

75

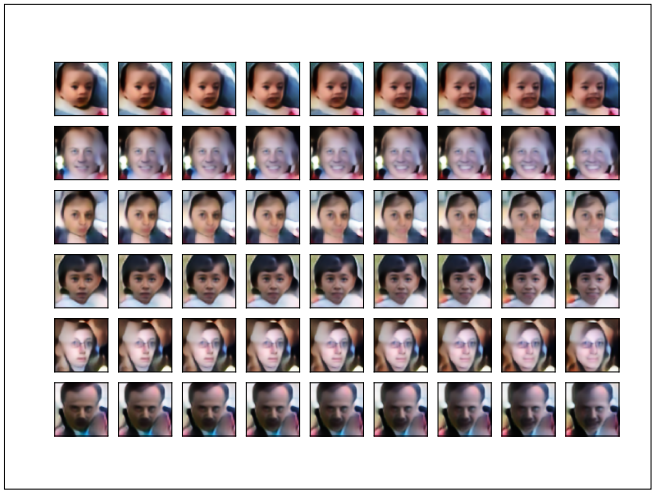
If we feed these points to the decoder, this is what we get. This is a decent enough way of turning an autoencoder into a generative model. But it's not very *principled*: we're training for reconstruction error, and then turning the result into an autoencoder. Is there a way to train for **maximum likelihood** directly?



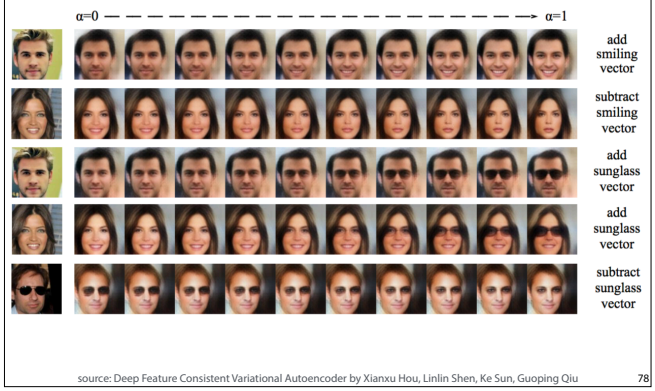
ae vae

generated

76



with VAEs

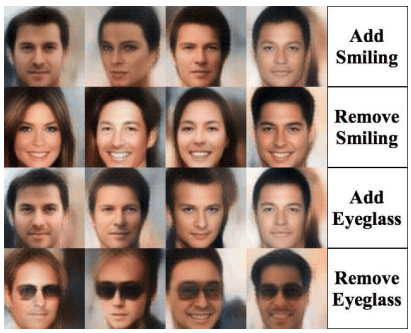


source: Deep Feature Consistent Variational Autoencoder by Xianxu Hou, Linlin Shen, Ke Sun, Guoping Qiu 78

Here are some examples from a more elaborate VAE

source: Deep Feature Consistent Variational Autoencoder by Xianxu Hou, Linlin Shen, Ke Sun, Guoping Qiu

with VAEs



source: **https://houxianxu.github.io/assets/project/dfcvae** 79

source: **https://houxianxu.github.io/assets/project/dfcvae**

interpolation

AE

i went to the store to buy some groceries .
i store to buy some groceries .
i were to buy any groceries .
horses are to buy any groceries .
horses are to buy any animal .
horses the favorite any animal .
horses the favorite favorite animal .
horses are my favorite animal .

VAE

he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .

80

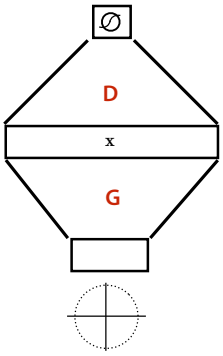
While the added value of the VAE is a bit difficult to detect, in other domains it's more clear.

Here is an example of interpolation on sentences. First using a regular autoencoder, and then using a VAE. Note that the intermediate sentences for the AE are non-grammatical, but the intermediate sentences for the VAE are all grammatical.

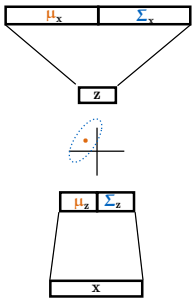
source: Generating Sentences from a Continuous Space by Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, Samy Bengio
https://arxiv.org/abs/1511.06349

summary: generative modeling

GANs



VAEs



81

GANs and VAEs

Allow us to train generator networks, avoiding mode collapse

GANs	VAEs
Better for images, often poor in other domains.	Work for language, music, etc.
Ad-hoc model, difficult to establish what is being optimized.	Derived from first principles.
Can't handle discrete data easily.	Allow mapping from data to latent space.
	Can't handle discrete latent variables easily.

82

VAEs and PCA

Mapping to low-dimensional *whitened* latent space (zero, mean, decorrelated).

PCA	VAEs
Linear transformation	Nonlinear transformation
Analytical solution	GD required
Principal components often meaningful, ordered by impact.	Latent dimensions not usually meaningful.

83

mlcourse@peterbloem.nl

We see that GANs are in many ways the inverse of autoencoders, in that GANS have the data space as the inside of the network, and VAEs have it as the outside.